Creating forms from models

**ModelForm**

***class ModelForm***

If you’re building a database-driven app, chances are you’ll have forms that map closely to Django models.

For instance, you might have a **BlogComment** model, and you want to create a form that lets people submit

comments. In this case, it would be redundant to define the field types in your form, because you’ve already

defined the fields in your model.

For this reason, Django provides a helper class that lets you create a Form class from a Django model.

For example:>>> from django.forms import ModelForm

>>> from myapp.models import Article

# Create the form class.

>>> class ArticleForm(ModelForm):

... class Meta:

... model = Article

... fields = ["pub\_date", "headline", "content", "reporter"]

...

# Creating a form to add an article.

>>> form = ArticleForm()

# Creating a form to change an existing article.

>>> article = Article.objects.get(pk=1)

>>> form = ArticleForm(instance=article)

**Field types**

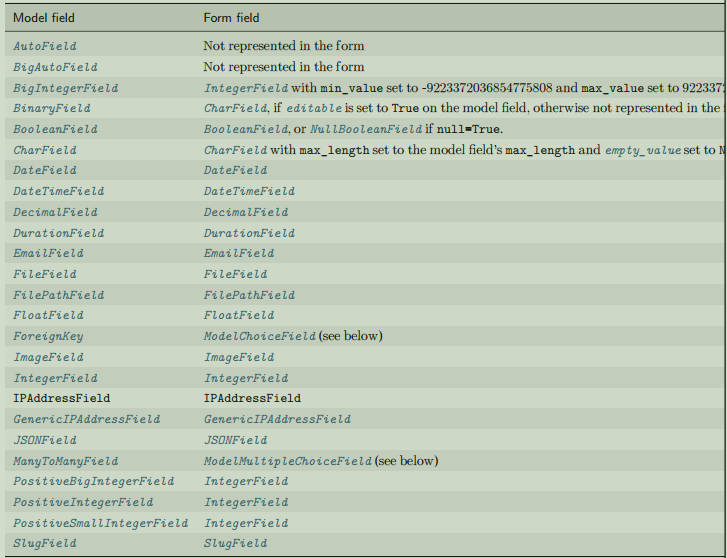
The generated Form class will have a form field for every model field specified, in the order specified in the

fields attribute.

Each model field has a corresponding default form field. For example, a **CharField** on a model is represented

as a CharField on a form. A model **ManyToManyField** is represented as a **MultipleChoiceField**. Here is the

full list of conversions:



A screenshot of a computer

Description automatically generated

As you might expect, the **ForeignKey** and **ManyToManyField** model field types are special cases:

• **ForeignKey** is represented by django.forms.ModelChoiceField, which is a ChoiceField whose

choices are a model QuerySet.

• **ManyToManyField** is represented by django.forms.ModelMultipleChoiceField, which is a

**MultipleChoiceField** whose choices are a model QuerySet.

In addition, each generated form field has attributes set as follows:

• If the model field has blank=True, then required is set to False on the form field. Otherwise,

required=True.

• The form field’s label is set to the verbose\_name of the model field, with the first character capitalized.

• The form field’s help\_text is set to the help\_text of the model field.

• If the model field has choices set, then the form field’s widget will be set to Select, with choices coming

from the model field’s choices. The choices will normally include the blank choice which is selected

by default. If the field is required, this forces the user to select. The blank choice will not be

included if the model field has blank=False and an explicit default value (the default value will be

initially selected instead).

Finally, note that you can override the form field used for a given model field. See Overriding the default

fields below.A full example

Consider this set of models:

from django.db import models

from django.forms import ModelForm

TITLE\_CHOICES = {

"MR": "Mr.",

"MRS": "Mrs.", "MS": "Ms.",

}

class Author(models.Model):

name = models.CharField(max\_length=100)

title = models.CharField(max\_length=3, choices=TITLE\_CHOICES)

birth\_date = models.DateField(blank=True, null=True)

def \_\_str\_\_(self):

return self.name

class Book(models.Model):

name = models.CharField(max\_length=100)

authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):

class Meta:

model = Author

fields = ["name", "title", "birth\_date"]

class BookForm(ModelForm):

class Meta:

model = Book

fields = ["name", "authors"]

With these models, the ModelForm subclasses above would be roughly equivalent to this (the only difference

being the save() method, which we’ll discuss in a moment.):

from django import forms

class AuthorForm(forms.Form):

name = forms.CharField(max\_length=100)

title = forms.CharField(

max\_length=3,

widget=forms.Select(choices=TITLE\_CHOICES),

)

birth\_date = forms.DateField(required=False)

class BookForm(forms.Form):

name = forms.CharField(max\_length=100)

authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())

Validation on a ModelForm

There are two main steps involved in validating a **ModelForm**:

1. Validating the form

2. Validating the model instance

Just like normal form validation, model form validation is triggered implicitly when calling **is\_valid()** or accessing the errors attribute and explicitly when calling **full\_clean(),** although you will typically not use the latter method in practice.

Model validation (**Model.full\_clean())** is triggered from within the form validation step, right after the

form’s **clean()** method is called.

Warning: The cleaning process modifies the model instance passed to the ModelForm constructor in various ways. For instance, any date fields on the model are converted into actual date objects. Failed validation may leave the underlying model instance in an inconsistent state and therefore it’s not recommended to reuse it.

# Overriding the clean() method

You can override the clean() method on a model form to provide additional validation in the same way you

can on a normal form.

A model form instance attached to a model object will contain an instance attribute that gives its methods

access to that specific model instance.

Warning: The **ModelForm.clean()** method sets a flag that makes the model validation step validate the uniqueness of model fields that are marked as unique**, unique\_together** or **unique\_for\_date|month|year**.

If you would like to override the clean() method and maintain this validation, you must call the parent

class’s **clean()** method.

Interaction with model validation

As part of the validation process, ModelForm will call the clean() method of each field on your model that has a corresponding field on your form. If you have excluded any model fields, validation will not be run on those fields.

The model’s clean() method will be called before any uniqueness checks are made.

### Considerations regarding model’s error\_messages

Error messages defined at the **form field** level or at the **form Meta** level always take precedence over the

error messages defined at the **model field** level.

Error messages defined on model fields are only used when the **ValidationError** is raised during the model

validation step and no corresponding error messages are defined at the form level.

You can override the error messages from **NON\_FIELD\_ERRORS** raised by model validation by adding the

**NON\_FIELD\_ERRORS** key to the **error\_messages** dictionary of the ModelF**orm’**s inner **Meta** class:

from django.core.exceptions import NON\_FIELD\_ERRORS

from django.forms import ModelForm

class ArticleForm(ModelForm):

class Meta:

error\_messages = {

NON\_FIELD\_ERRORS: {

"unique\_together": "%(model\_name)s's %(field\_labels)s are not unique.",

}

}

### The save() method

Every ModelForm also has a save() method. This method creates and saves a database object from the data

bound to the form. A subclass of ModelForm can accept an existing model instance as the keyword argument

instance; if this is supplied, save() will update that instance. If it’s not supplied, save() will create a new

instance of the specified model:

>>> from myapp.models import Article

>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.

>>> new\_article = f.save()

# Create a form to edit an existing Article, but use

# POST data to populate the form.

>>> a = Article.objects.get(pk=1)

>>> f = ArticleForm(request.POST, instance=a)

>>> f.save()

Note that if the form hasn’t been validated, calling save() will do so by checking **form.errors**. A **ValueError**

will be raised if the data in the form doesn’t validate – i.e., **if form.errors** evaluates to **True**.

If an optional field doesn’t appear in the form’s data, the resulting model instance uses the model field

default, if there is one, for that field. This behavior doesn’t apply to fields that use **CheckboxInput**,

**CheckboxSelectMultiple**, or **SelectMultiple** (or any custom widget whose **value\_omitted\_from\_data()**

method always returns **False**) since an unchecked checkbox and unselected **<select multiple>** don’t appear in the data of an **HTML** form submission. Use a custom form field or widget if you’re designing an **API**

and want the default fallback behavior for a field that uses one of these widgets.

This **save()** method accepts an optional commit keyword argument, which accepts either **True** or **False**. If

you call **save()** with **commit=False**, then it will return an object that hasn’t yet been saved to the database.

In this case, it’s up to you to call **save()** on the resulting model instance. This is useful if you want to do

custom processing on the object before saving it, or if you want to use one of the specialized model saving

options. commit is **True** by default.

Another side effect of using **commit=False** is seen when your model has a many-to-many relation with another **model**. If your model has a **many-to-many relation** and you specify **commit=Fals**e when you save a

form, **Django** cannot immediately save the form data for the **many-to-many relation**. This is because it isn’t

possible to save **many-to-many** data for an instance until the instance exists in the database.

To work around this problem, every time you save a form using **commit=False**, Django adds a **save\_m2m()**

method to your **ModelForm** subclass. After you’ve manually saved the instance produced by the **form**, you

can invoke **save\_m2m()** to save the **many-to-many** form data. For example:

# Create a form instance with POST data.

>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.

>>> new\_author = f.save(commit=False)

# Modify the author in some way.

>>> new\_author.some\_field = "some\_value"# Save the new instance.

>>> new\_author.save()

# Now, save the many-to-many data for the form.

>>> f.save\_m2m()

Calling **save\_m2m()** is only required if you use **save(commit=False**). When you use a save**()** on a form, all data – including **many-to-many** data – is saved without the need for any additional method calls. For example:

# Create a form instance with POST data.

>>> a = Author()

>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do anything else.

>>> new\_author = f.save()

Other than the **save()** and **save\_m2m()** methods, a **ModelForm** works exactly the same way as any other

forms form. For example, the **is\_valid()** method is used to check for validity, the **is\_multipart()** method

is used to determine whether a form requires multipart file upload (and hence whether **request.FILES** must

be passed to the form), etc.

### Selecting the fields to use

It is strongly recommended that you explicitly set all fields that should be edited in the form using the fields

attribute. Failure to do so can easily lead to security problems when a form unexpectedly allows a user to set

certain fields, especially when new fields are added to a model. Depending on how the form is rendered, the

problem may not even be visible on the web page.

The alternative approach would be to include all fields automatically or remove only some. This fundamental approach is known to be much less secure and has led to serious exploitation on major websites (e.g. GitHub).

There are, however, two shortcuts available for cases where you can guarantee these security concerns do

not apply to you:

1. Set the **fields** attribute to the special value '**\_\_all\_\_'** to indicate that all fields in the model should be used. For example:

from django.forms import ModelForm

class AuthorForm(ModelForm):

class Meta: model = Author

fields = "\_\_all\_\_"

2. Set the **exclude** attribute of the ModelForm’s inner Meta class to a list of fields to be excluded from the form.

For example:

class PartialAuthorForm(ModelForm):

class Meta:

model = Author

exclude = ["title"]

Since the Author model has the 3 fields **name**, **title** and **birth\_date**, this will result in the fields name and **birth\_date** being present on the form.If either of these are used, the order the fields appear in the form will be the order the fields are defined in the

model, with **ManyToManyField** instances appearing last.

In addition, **Django** applies the following rule: if you set **editable=False** on the model field, any form created

from the model via **ModelForm** will not include that field.

Note: Any fields not included in a form by the above logic will not be set by the form’s save() method.

Also, if you manually add the excluded fields back to the form, they will not be initialized from the model

instance.

Django will prevent any attempt to save an incomplete model, so if the model does not allow the missing fields

to be empty, and does not provide a default value for the missing fields, any attempt to save() a ModelForm

with missing fields will fail. To avoid this failure, you must instantiate your model with initial values for the

missing, but required fields:

author = Author(title="Mr")

form = PartialAuthorForm(request.POST, instance=author)

form.save()

Alternatively, you can use save(commit=False) and manually set any extra required fields:

form = PartialAuthorForm(request.POST)

author = form.save(commit=False)

author.title = "Mr"

author. Save()

### Overriding the default fields

The default field types, as described in the **Field types** table above, are sensible defaults. If you have a **DateField** in your model, chances are you’d want that to be represented as a **DateField** in your form. But **ModelForm** gives you the flexibility of changing the form field for a given model. To specify a custom widget for a field, use the widgets attribute of the inner **Meta class**. This should be a dictionary mapping field names to widget classes or instances.

For example, if you want the **CharField** for the name attribute of Author to be represented by a **<textarea>** instead of its default **<input type="text">**, you can override the field’s widget:

from django.forms import ModelForm, Textarea

from myapp.models import Author

class AuthorForm(ModelForm):

class Meta:

model = Author

fields = ["name", "title", "birth\_date"]

widgets = {

"name": Textarea(attrs={"cols": 80, "rows": 20}),

}

The widgets dictionary accepts either widget instances (e.g., Textarea(...)) or classes (e.g., Textarea). Note that the widgets dictionary is ignored for a model field with a non-empty choices attribute. In this case, you must override the form field to use a different widget.

Similarly, you can specify the labels, help\_texts and error\_messages attributes of the inner Meta class if you want to further customize a field.

For example if you wanted to customize the wording of all user facing strings for the name field:

from django.utils.translation import gettext\_lazy as \_

class AuthorForm(ModelForm):

class Meta:

model = Author

fields = ["name", "title", "birth\_date"]

labels = {

"name": \_("Writer"),

}

help\_texts = {

"name": \_("Some useful help text."),

}

error\_messages = {

"name": {

"max\_length": \_("This writer's name is too long."),

},

}

You can also specify **field\_classes** or **formfield\_callback** to customize the type of fields instantiated by

the form.

For example, if you wanted to use **MySlugFormField** for the slug field, you could do the following:

from django.forms import ModelForm

from myapp.models import Article

class ArticleForm(ModelForm):

class Meta:

model = Article

fields = ["pub\_date", "headline", "content", "reporter", "slug"]

field\_classes = {

"slug": MySlugFormField,

}or:

from django.forms import ModelForm

from myapp.models import Article

def formfield\_for\_dbfield(db\_field, \*\*kwargs):

if db\_field.name == "slug":

return MySlugFormField()

return db\_field.formfield(\*\*kwargs)

class ArticleForm(ModelForm):

class Meta:

model = Article

fields = ["pub\_date", "headline", "content", "reporter", "slug"]

formfield\_callback = formfield\_for\_dbfield

Finally, if you want complete control over a field – including its **type, validators, required, etc**. – you can do this by declaratively specifying fields like you would in a regular Form.

If you want to specify a field’s validators, you can do so by defining the field declaratively and setting its validators parameter:from django.forms import CharField, ModelForm

from myapp.models import Article

class ArticleForm(ModelForm):

slug = CharField(validators=[validate\_slug])

class Meta:

model = Article

fields = ["pub\_date", "headline", "content", "reporter", "slug"]

Note: When you explicitly instantiate a form field like this, it is important to understand how ModelForm

and regular Form are related.

ModelForm is a regular Form which can automatically generate certain fields. The fields that are automatically

generated depend on the content of the Meta class and on which fields have already been defined declaratively.

Basically, ModelForm will only generate fields that are missing from the form, or in other words, fields that

weren’t defined declaratively.

Fields defined declaratively are left as-is, therefore any customizations made to Meta attributes such as

widgets, labels, help\_texts, or error\_messages are ignored; these only apply to fields that are generated

automatically.

Similarly, fields defined declaratively do not draw their attributes like max\_length or required from the

corresponding model. If you want to maintain the behavior specified in the model, you must set the relevant

arguments explicitly when declaring the form field.

For example, if the Article model looks like this:

class Article(models.Model):

headline = models.CharField(

max\_length=200,

null=True,

blank=True,

help\_text="Use puns liberally",

)

Content = models.TextField()

and you want to do some custom validation for headline, while keeping the blank and help\_text values as

specified, you might define ArticleForm like this:

class ArticleForm(ModelForm):

headline = MyFormField( max\_length=200,

required=False,

help\_text="Use puns liberally",

)

class Meta:

model = Article

fields = ["headline", "content"]You must ensure that the type of the form field can be used to set the contents of the corresponding model field. When they are not compatible, you will get a ValueError as no implicit conversion takes place.

### Enabling localization of fields

By default, the fields in a ModelForm will not localize their data. To enable localization for fields, you can use

the localized\_fields attribute on the Meta class.

>>> from django.forms import ModelForm

>>> from myapp.models import Author

>>> class AuthorForm(ModelForm):

...

... class Meta:

... model = Author

... localized\_fields = ['birth\_date']

If localized\_fields is set to the special value '\_\_all\_\_', all fields will be localized.

### Form inheritance

As with basic forms, you can extend and reuse ModelForms by inheriting them. This is useful if you need to

declare extra fields or extra methods on a parent class for use in a number of forms derived from models. For

example, using the previous ArticleForm class:

>>> class EnhancedArticleForm(ArticleForm):

... def clean\_pub\_date(self): ...

...

This creates a form that behaves identically to ArticleForm, except there’s some extra validation and cleaning for the **pub\_date** field. You can also subclass the parent’s Meta inner class if you want to change the **Meta.fields** or **Meta.exclude** lists:

>>> class RestrictedArticleForm(EnhancedArticleForm):

... class Meta(ArticleForm.Meta):

... exclude = ["body"]

...

This adds the extra method from the **EnhancedArticleForm** and modifies the original **ArticleForm.Meta** to

remove one field.

There are a couple of things to note, however.

• Normal Python name resolution rules apply. If you have multiple base classes that declare a Meta inner class, only the first one will be used. This means the child’s **Meta**, if it exists, otherwise the Meta of the first parent, etc.

• It’s possible to inherit from both Form and ModelForm simultaneously, however, you must ensure that ModelForm appears first in the MRO. This is because these classes rely on different **metaclasses** and a class can only have one **metaclass**.

• It’s possible to declaratively remove a Field inherited from a parent class by setting the name to be

None on the subclass.

You can only use this technique to opt out from a field defined declaratively by a parent class; it won’t prevent the **ModelForm** **metaclass** from generating a default field. To opt-out from default fields, see Selecting the fields to use.

### Providing initial values

As with regular forms, it’s possible to specify initial data for forms by specifying an initial parameter when

instantiating the form. Initial values provided this way will override both initial values from the form field

and values from an attached model instance. For example:

>>> article = Article.objects.get(pk=1)

>>> article.headline

'My headline'

>>> form = ArticleForm(initial={"headline": "Initial headline"}, instance=article)

>>> form["headline"].value()

'Initial headline'

## ModelForm factory function

You can create forms from a given model using the standalone function modelform\_factory(), instead of

using a class definition. This may be more convenient if you do not have many customizations to make:

>>> from django.forms import modelform\_factory

>>> from myapp.models import Book

>>> BookForm = modelform\_factory(Book, fields=["author", "title"])

This can also be used to make modifications to existing forms, for example by specifying the widgets to be

used for a given field:

>>> from django.forms import Textarea

>>> Form = modelform\_factory(Book, form=BookForm, widgets={"title": Textarea()})

The fields to include can be specified using the fields and exclude keyword arguments, or the corresponding attributes to the ModelForm inner Meta class. Please see the ModelForm Selecting the fields to use documentation.

. . . or enable localization for specific fields:

>>> Form = modelform\_factory(Author, form=AuthorForm, localized\_fields=["birth\_date"])

---------------------------- END --------------------------